DRL-Epanet: Deep reinforcement learning for optimal control at scale in Water Distribution Systems

Anonymous Author(s)
Affiliation
Address
email

Abstract

Deep Reinforcement learning has known a revolution in recent years, it has allowed researchers to tackle a wide range of sequential decision problems that were inaccessible to previous methods. However, the use of this technique in Water Distribution Systems is still very shy. In this paper, we show that DRL can be coupled with the widely popular hydraulic simulator Epanet, and that DRL-Epanet can be used on a number of WDS problems that represent a challenge to current techniques. We take as a concrete example the problem of pressure control in WDS. We show that DRL-Epanet can scale to huge action spaces, and we demonstrate its effectiveness on a problem with more than 1 million possible actions at each time step. We also show that it can deal with uncertainty such as stochastic demands, contamination, or other risks, as an example, we take on the problem of pressure control in the presence of random pipe bursts. We show that the BDQ algorithm is able to learn in this setting and we improve it with an algorithmic modification BDQF (BDQ with Fixed actions) which achieves better rewards especially when allowed actions are sparse in the action space. Finally, we argue that DRL-Epanet can be used for real-time control in smart WDS, another advantage over current methods.

1 Introduction

Water is a finite resource serving an escalating number of users. In fact, about 1,5 billion people have risen in the global population in the last 20 years, increasing the demand for clean water. Moreover, urbanization, climate change, and drought have led to the over-exploitation of water resources. As a consequence, adopting more sustainable water management strategies is now a pressing matter for municipalities, water utility companies, and societies in general.

The complex and growing water networks challenge operators in achieving cost-effective, yet satisfactory operations. New techniques are being developed by researchers, in their history of optimization in WDS [1] authors subdivide the WDS optimization methods into two broad classes: deterministic, and stochastic (heuristic) optimization. Mainly, the deterministic methods consist of: Linear Programming (LP): they can find optimal solutions but only work for a continuous problem with a linear objective function subject to linear constraints. Dynamic Programming (DP): DP is another deterministic methodology suitable for multistage optimization problems, mostly used for pump scheduling. However, the authors note that this methodology suffers from the so-called 'curse of dimensionality', which limits to some extent its application to large WDS. Nonlinear Programming (NLP): similarly to LP, NLP works with continuous variables, it is limited in the number of variables and constraints it can handle and thus can only manage WDS of limited size.

The non-linearity as well as the discrete nature of many of the problems, has led researchers to move from deterministic methods towards heuristic optimization techniques [1] which are usually

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

coupled with hydraulic solvers such as Epanet. The main advantage of these over the deterministic optimization approaches is that they have found solutions to large and complex problems that no satisfactory problem-specific deterministic algorithm can solve. Various metaheuristics are applied to WDS optimization, including genetic algorithms and their numerous variants, simulated annealing, particle swarm optimization...

Both deterministic and stochastic optimization methods are further challenged in the real-time control setting, where it is necessary to determine the best set of actions from continuous measurements that are taken in real-time. In this case, a compromise between the efficiency and the precision of the method has to be negotiated, which translates to simplified hydraulic models and/or a very limited computing budget for the optimization procedure, and therefore affects the quality of the solution

In recent years, Deep Reinforcement Learning (DRL) has revolutionized the field of sequential decision-making. It has achieved breakthrough results in several domains such as superhuman game playing in Chess and Go, protein folding prediction, traffic lights control, self-driving, robotic control... DRL has many advantages over these classical methods that can be leveraged for solving optimal control problems in WDS: 1 - Scaling to high-dimensional problems, 2- Dealing with the stochasticity of the environment, 3- Suitable for RTC since it needs to do only inference after it has been trained, 4- Continuous learning after deployment.

In this paper, we'll present a method for pressure control optimization in WDS based on Deep Reinforcement Learning and the popular hydraulic software Epanet. We will demonstrate its effectiveness in real-world cases with large action spaces and in the presence of random pipe breaks. We will argue that this framework is general enough to tackle a wide range of sequential decision optimization problems in WDS and that it can be applied for real-time control in smart WDS.

2 Related works

In [2] Epanet was used alongside a genetic algorithm (GA) for optimizing the number of valves and their locations for pressure control in the WDS. In [3] authors developed an optimization procedure for energy recovery and reduction of leakage utilizing GA with the hydraulic modeling performed in Epanet. Ant colony optimization was used in [4] in conjunction with Epanet for optimal control of pumps in WDS, warnings issued by Epanet for the inefficient operation of pumps were used in the constraint handling procedure. However, the recent survey on control optimization in WDS [5] concludes that even with parallel programming techniques and more efficient deterministic optimization methods, WDS simulations may still be computationally prohibitive for Real-Time Control (RTC), especially as the fidelity of the model and the number of decision variables increase.

Mullapudi et al. (2020) [6] used reinforcement learning for real-time control of stormwater systems, they used the Deep Q-Network algorithm and limited the action space to only 27 possible actions. In water distribution systems, [7] used reinforcement learning for stochastic optimization of multi-reservoir systems, [8] Used DRL for real-time optimization of pumps in WDS, they found that their agent is capable of performing as well as the best conventional techniques while being $2 \times$ faster. They also noted the advantage of the DRL for RTC in comparison to previous methods. [9] Authors used DRL with a quadratic approximation of WDS hydraulics to solve pressure control using PRVs, their emphasis was on the "model-free" nature of their approach and their treatment of the DRL method used was minimal. To the best of our knowledge, these papers and others that applied DRL for WDS were restricted to small action spaces in order of dozens of possible actions at max.

3 Contributions

We present a heuristic method that couples the Epanet solver with Deep Reinforcement Learning for optimal control in real WDS. Authors in [8] demonstrated the effectiveness of this method for Real-Time Optimization of Pumps. We apply it here to pressure control and we emphasize its generality for other WDS optimal control problems for large action spaces and in a stochastic setting. We use a Reinforcement Learning algorithm based on branching neural network architecture (BDQ) for representing the Q-value function, which allows scaling to large action spaces and that can be used for discrete action values as it's the case for open/close valves and that of on/off pumps, for continuous actions and for mixed action spaces.

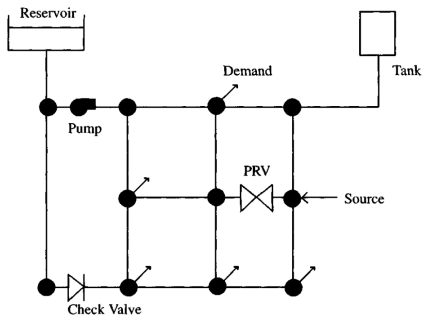


Figure 1: Node-link representation of a WDS

We demonstrate the effectiveness of this approach for pressure control in normal conditions, i.e. no pipe bursts, in nine real-world WDS. We show that we are able to reduce the average pressure by up to 26%. We also demonstrate the advantage of using BDQ over DQN on a high-dimensional action space with more than one million actions.

We then modify the BDQ algorithm to deal with random bursts of pipes in WDS. We introduce BDQF that improves over BDQ in this scenario, especially when allowable actions at each step are sparse in the action space. To the best of our knowledge, our approach is the first for pressure control in the presence of pipe bursts.

We use our own optimized OpenAI Gym [10] interface of the Epanet solver in python, and our own implementation of the BDQ algorithm that we made available as a contribution to Tianshou [11], an open source library under MIT license for reinforcement learning (look for Branching DQN).

4 Framing the WDS pressure control as an RL problem

Water distribution networks can be modeled as a collection of *links* connecting *nodes*. Water flows along links and enters or leaves the system at nodes. All the actual physical components of a distribution system can be represented in terms of these constructs. One particular scheme for accomplishing this is shown in 1. In it, links consist of pipes, pumps, or control valves. Pipes convey water from one point to another, pumps raise the hydraulic head of water, and control valves maintain specific pressure or flow conditions. Nodes consist of pipe junctions, reservoirs, and tanks. Junctions are demand nodes where links connect and where water consumption occurs. Reservoir nodes represent fixed-head boundaries, such as lakes, groundwater aquifers, treatment plant clear wells, or connections to parts of a system not being modeled. Tanks are storage facilities, the volume, and water level of which can change over an extended period of system operation.

We want to train a reinforcement learning agent to control valves in the WDS to minimize the overall pressure in the network under the constraints of minimum and maximum pressure, and without emptying or overfilling the tanks.

We take a time-step of one hour, which is the usual time step for hydraulic simulations in WDS. We consider a full-length episode to be 24 hours, which corresponds to the usual cycle of WDSs operations. At each time step the agent receives the pressures at demand nodes, the tank levels, and the time, this is the state s of the environment. It should then decide the opening values of the valves, these are the actions a. A simulation is then performed via Epanet to calculate the new pressures at the demand nodes and the new tank levels: the environment transitions to the new state s'. The reward r is then calculated and returned to the agent. The quadruple $e = (s, a, r, s')$ is called an experience 2.

The rewards

Rewards depend on two terms:

Pressures: Our goal is to insure service for the highest number of clients while maintaining operational constraints. Hence, we use a cost function c_p for counting the nodes that are served with

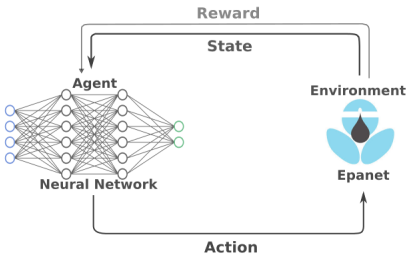


Figure 2: WDS reinforcement learning setup

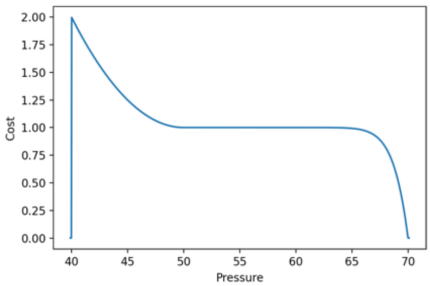


Figure 3: Cost function with smooth transitions.

pressures inside the required range. c_p is equal to a number between one and two for nodes with pressures between p_{min} and p_{max} and to zero otherwise, it is biased towards smaller pressures to incentivize pressure reduction, see Figure 3. These values are then summed and normalized over the number of nodes.

Tanks: Over-topping or emptying the tanks are considered to be episode-ending incidents.

The majority of water-distribution networks operate on a 24 h cycle in which the storage tanks are refilled overnight when the charge for electricity is low and then drawn down during the daytime hours when demands are high. This not only reduces the operating costs but also ensures a turnover of the water in storage, thereby avoiding stagnation. It is of most importance to avoid over-topping or emptying the tanks, these events are considered to be episode-ending incidents.

Therefore the reward at time-step t is formalized as:

$$r_t = \sum_{i=1}^{N} \frac{c_p(p_i)}{n}$$

Where p_i is the pressure at demand node i, N is the number of nodes in the WDS.

The goal of the RL agent is to maximize the cumulative discounted reward over the episode:

$$G = \sum_{t=0}^{t=t_f} \gamma^t r_t$$

Where t_f is equal to 23 for a non interrupted episode, otherwise it is the time when the incident of over-topping or emptying any of the tanks occurs.

5 DRL algorithms used

DQN and BDQ

DQN is a classical value-based Deep Reinforcement learning algorithm, it was introduced by Deepmind in 2015 kick-starting the modern revolution in RL, and has been used since then in numerous publications. However, DQN suffers from the curse of dimensionality, as the network representing the Q-values scales exponentially w.r.t. the number of dimensions in the action space. BDQ [12] was introduced to solve this problem by adopting a branching architecture that represents each dimension in the action space as a branch, and has a common network "trunk" that coordinates between the branches, see Annex A and B for a detailed presentation of the DQN and BDQ algorithms.

Adapting BDQ for the pipe failure scenario

In the case of pipe failure, some of the PRVs need to be closed for the rest of the episode in order to isolate the failure sector. In this case, the corresponding actions are no longer available for the agent to freely modify. The agent might have 16 valves to control in one state and 13 valves in another state.

Like previous work [13] [14] [15], to formulate this as an RL problem with a fixed set action space we consider the full action space containing all the possible actions and we use action masking to consider only valid actions at a given state. [16] presents a theoretical analysis of action masking in policy gradient algorithms. All of these works consider action masking for policy gradient algorithms, we didn't find any scientific publications that consider it for value-based algorithms. However, we found masking for value-based algorithms discussed in some forums and a blog [17], and we found it also in the DQN implementation in Tianshou [11]. These assign small or negative Q-values for masked actions such as they are never selected by the policy $\pi(a|s) = \underset{a}{\operatorname{argmax}}\ Q(s,a)$. In this section, we will perform a theoretical analysis of action masking for value-based algorithms in the case of multidimensional action space where masking means fixing the values along some dimensions in the action space.

Fixing some actions still produces a valid policy For a given pipe failure lets denote the subset of valves that need to be closed by V_f, and let A_f be the elements of V_f that are in the action space, and let A_{nf} be the rest of the action space. Then, we can represent the agent's actions as: $a = (a_{nf}, a_f)$ with $a_{nf} \in A_{nf}, a_f \in A_f$. The important point here is: when action a is taken, only the a_{nf} part has consequences on the environment (since a_f are fixed and even if changed by the agent, the environment will ignore the change). Therefore, $r(s,a) = r(s,a_{nf})$ and it follows that $Q(s,a) = Q(s,a_{nf})$ which means that, at least for the tabular case, learning $Q(s,a)$ is equivalent to learning $Q(s,a_{nf})$, consequently, any Q-learning algorithm can produce a valid policy without modification. However, learning $Q(s,a)$ is less efficient, since it is a bigger table and it will have unnecessary duplicates since $Q(s,(a_{nf},a_f)) = Q(s,(a_{nf},a'_f))$ for any a_f, a'_f. Moreover, in the function approximation case, this inefficiency will be manifested as a learning noise corresponding to assigning rewards to actions that have no effects.

The advantage of separating the action dimensions in BDQ can be leveraged to mitigate the aforementioned problems. We propose "freezing" the action dimensions that have fixed values by preventing the back-propagation of the learning signal in branches where actions are fixed. Algorithmically we modify the temporal-difference target 9 and the loss function 8 of BDQ as follows:

$$y = r + \gamma\ \frac{1}{N_{nf}} \sum_{d_{nf}} Q'_d(s,\ \underset{a' \in \mathcal{A}_{d_{nf}}}{\operatorname{argmax}}\ Q_\theta(s',a')) \tag{1}$$

$$L = E_{(s,a,r,s')\sim\mathcal{D}}\big[\sum_{d_{nf}} \big(y_{d_{nf}} - Q_{d_{nf}}(s, a_{d_{nf}})\big)^2\big] \tag{2}$$

Where N_{nf} and d_{nf} are respectively the number and the dimensions of the non-fixed part of the action space. The detailed algorithm can be found in Annex 1

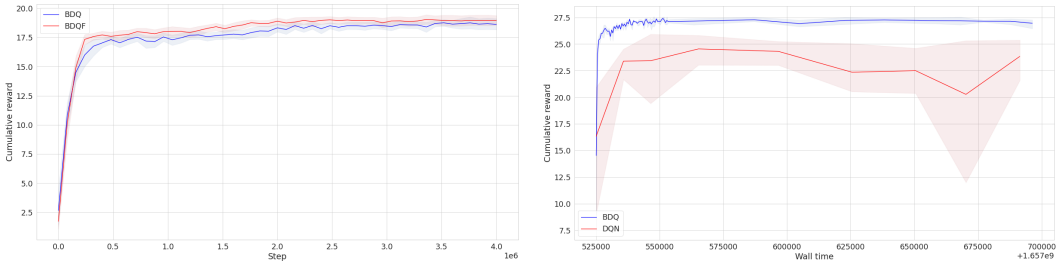
6 Results

Optimal pressure control in normal conditions

We run both DQN and BDQ algorithms on a set of nine real WDS, eight of which are of Barcelona sectors, and one in Marbella. We run in normal conditions, i.e. without any pipe burst incidents. The number of control valves for these networks is either two or four, and the valves can be set to any value between 10 and 50 PCA. We discretize the action space by dividing the interval 10-50pm into eight equal bins.

In all the WDS the agents were able to find better solutions than the reference that is actually used by operators, an improvement of up to 26% in mean pressure was achieved, with a mean of 13% of improvement over all the WDS, see figure 5 for detailed improvement in pressure per WDS.

Table C.2 contains the description of each of the networks as well as the mean pressures before and after optimization using Deep RL. In **bold** are the cases where BDQ achieved better rewards than DQN. In all cases the best cumulative reward achieved by both algorithms was almost the same with at most 0.5% of difference, this is normal since the action space for these networks is relatively low at $8^4 = 256$ possible actions at max.



(a) Comparison of BDQ and DQN on the 8 WDS in table C.2 with bootstrapped 95% CI

(b) BDQ vs DQN on the Gracia Nova WDS with 32 bins for each of the 4 valves

Figure 4: BDQ vs DQN, 8 bins and 32 bins

In 4a we plot a summary of the learning curves of BDQ and DQN over all the nine WDS. A total of 25 randomly seeded runs were performed with each taking 48h of run-time. To summarize the plots we normalized the rewards and performed bootstrapped confidence interval, following the recommendations from [18]. Even though the best-achieved rewards at the test time are very similar, we notice in the plot that generally BDQ learns faster and is more stable than DQN.

When we consider 32 bins in each action dimension for the case of four valves, we get an action space size of $32^4 = 1\,048\,576$. In this case, the performance of DQN drops significantly compared to BDQ 4b, and if we go beyond this number to 50 bins, for example, the 16 Gb Nvidia Tesla v100 GPU that we use could no longer fit the DQN network in its memory. This is due to the way DQN networks represent the Q values, with one neuron in the output layer for each possible action, which makes it scale exponentially with the number of dimensions in action space. On the other hand, BDQ uses a branching architecture that allows it to scale linearly with the number of action space dimensions. This translates into a $7\times$ better performance for BDQ over DQN for the 32 bins case, where BDQ is able to perform 7M steps while DQN only does 1M with the same computation budget. The performance gap will grow exponentially as we add more control elements in the WDS, which means that only BDQ can be used for more complicated cases.

WDS Name	Nodes	Edges	Valves	$\lvert A \rvert$	Before	After	Improvement
Maternitat	1902	1964	2	16	59.9	44	26%
Placa Maragall	2370	2463	2	16	55.8	**46.7**	**16%**
Gran de Gracia	2756	2869	2	16	52.1	46.8	10%
Gracia Nova	3153	3274	4	256	55.1	**49.2**	**11%**
Placa Barraquer	1161	1194	2	16	57.6	**49.6**	**14%**
La Prosperitat	1841	1900	2	16	48.9	**47.3**	**3%**
Tuset	1328	1373	2	16	58	47	19%
Marbella	2418	2552	4	256	34.4	**30.1**	**12%**
Virrei Amat	2275	2364	2	16	51.2	**47.9**	**7%**

Optimal pressure control in the presence of random pipe bursts incidents

We consider the scenario where random leaks can happen in the network. The WDS, in this case, is that of the Castelldefels municipality, it has 2565 nodes, 2936 links, and 18 open/close control valves that can be manipulated by the agent to stabilize the pressure. The WDS is divided into 10 sectors that can be isolated in case of a pipe burst in any of them. When isolating a sector some of the control valves need to remain closed and can't be changed by the agent. As we mentioned earlier in 5 we can use BDQ without any modifications with some drawbacks, we can also our modified version that takes into account fixed actions BDQF. We train the two agents for 20 randomly seeded runs, each run spanning 48h. When training, at each episode we consider a random pipe burst which will correspond to isolating one of the 10 sectors. The agents are then evaluated every 10 epochs for each of the 10 possible pipe burst sectors, and the mean reward over these evaluations is calculated.

In this case, there is no reference solution to compare with the performance of the agents, however, we can see a clear learning tendency for both algorithms where both BDQ and BDQF were successful at learning good policies that are able to prevent early termination of the episodes due to emptying/over-

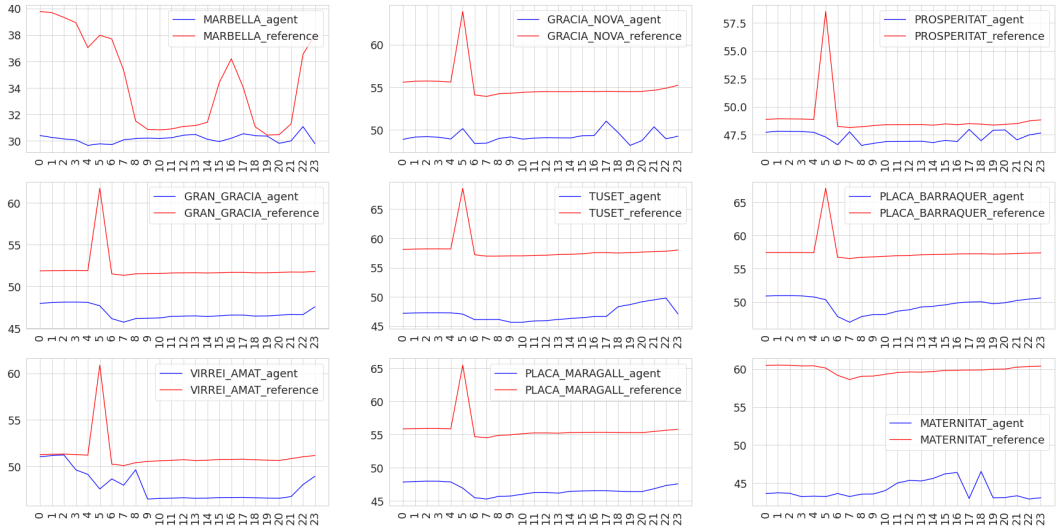
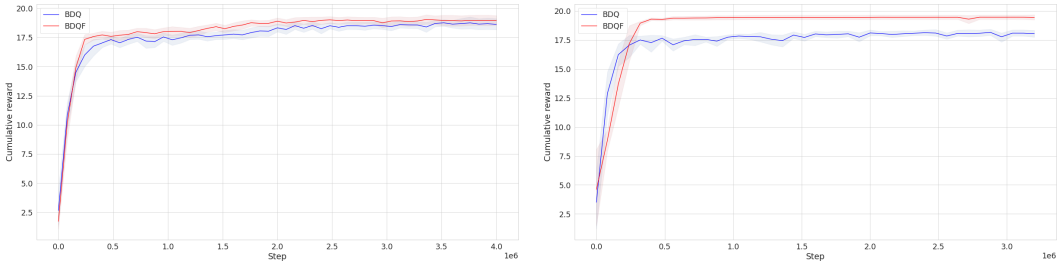


Figure 5: Mean Pressures comparison between the reference solution and the RL agent's solution

filling the tanks and achieve high rewards. At their best runs, BDQF achieved a mean reward over the 10 sectors of 20.2 while BDQ achieved 19.3. To analyze the performance of both algorithms we plot the bootstrapped 95% CI of the 10 sectors mean reward over 20 runs in 6a, we notice that BDQF performs better than BDQ. This is more apparent in 6b where we modified the action space to have sparse allowed actions, that are indeed controlled by the agent, this is done by adding dummy valves to the agent's action space that are in fact inconsequential in the environment.



(a) BDQ vs BDQF for pressure control in presence of random leaks

(b) BDQ vs BDQF for pressure control in presence of random leaks with sparse free actions

Figure 6: BDQ vs DQF

7 Conclusion and future work

We presented DRL-Epanet a method for optimal pressure control in water distribution systems (WDS) based on Deep Reinforcement Learning and Epanet. We demonstrated the effectiveness of this approach on ten real-world WDS, in all of which the A.I. solutions were superior to the reference solutions used by the operators by up to 26% of improvement in the mean pressure. We demonstrated the advantage of using BDQ over DQN for large action spaces. We then showed that BDQ can be used for optimal pressure control in presence of random pipe bursts, and we made an improvement BDQF over BDQ that mitigated the problem of learning noise introduced by considering gradients of actions that have no effect on the environment.

DRL-Epanet framework can be used to tackle a wide range of sequential decision optimization problems in WDS, such as pressure control as we demonstrated in this paper, pump optimization, energy, water quality, etc... or any combination of the aforementioned problems. We believe that any WDS sequential decision problem that can be simulated using Epanet, can in principle be solved in this framework. Moreover, DRL-Epanet provides a Real-Time Control solution, since once it

is trained, the agent is capable to react in real-time to changes in the network with only inference calculation needed, usually in the order of milliseconds, i.e. without recalculating the solution which can be a limiting factor for other methods. We showed in this paper that DRL-Epanet can deal with the stochasticity of the WDS environment in the form of random pipe bursts. Likewise, the DRL-Epanet agent can be trained with more stochastic scenarios to deal with demand uncertainty, contamination, and other component failures scenarios...

In the future, we would like to use model-based DRL to have better sample efficiency. In fact, previous work has shown that Artificial Neural Networks can be used as a substitute to Epanet simulation for WDS with a high degree of accuracy [19], which indicates that model-based DRL can be expected to work quite well for WDS. Another direction for research is using Graph Neural Networks which are the most natural representation for a WDS and can constitute a strong induction bias for DRL that might help in learning more efficiently and lead to better solutions, [20] have shown that such approach works for supervised learning for reconstructing nodal pressure in WDS. We would like also to be able to quantify the uncertainty of the agent actions in production, so we can differ control to the operator in uncharted territory, where the agent is uncertain.

References

[1] DA Savić, Helena Mala-Jetmarova, et al. History of optimization in water distribution system analysis. In *WDSA/CCWI Joint Conference Proceedings*, volume 1, 2018.

[2] LS Araujo, H Ramos, and ST Coelho. Pressure control for leakage minimisation in water distribution systems management. *Water resources management*, 20(1):133–149, 2006.

[3] Gideon Johannes Bonthuys, Marco Van Dijk, and Giovanna Cavazzini. Energy recovery and leakage-reduction optimization of water distribution systems using hydro turbines. *Journal of Water Resources Planning and Management*, 146(5):04020026, 2020.

[4] Manuel López-Ibáñez, T Devi Prasad, and Ben Paechter. Ant colony optimization for optimal control of pumps in water distribution networks. *Journal of water resources planning and management*, 134(4):337–346, 2008.

[5] Helena Mala-Jetmarova, Nargiz Sultanova, and Dragan Savic. Lost in optimisation of water distribution systems? a literature review of system operation. *Environmental Modelling Software*, 93:209–254, 2017.

[6] Abhiram Mullapudi, Matthew J. Lewis, Cyndee L. Gruden, and Branko Kerkez. Deep reinforcement learning for the real time control of stormwater systems. *Advances in Water Resources*, 140:103600, 2020.

[7] Jin-Hee Lee and John W. Labadie. Stochastic optimization of multireservoir systems via reinforcement learning. *Water Resources Research*, 43(11), 2007.

[8] Gergely Hajgató, György Paál, and Bálint Gyires-Tóth. Deep reinforcement learning for real-time optimization of pumps in water distribution systems. *Journal of Water Resources Planning and Management*, 146(11):04020079, 2020.

[9] Thapelo C. Mosetlhe, Yskandar Hamam, Shengzhi Du, Eric Monacelli, and Adedayo A. Yusuff. Towards model-free pressure control in water distribution networks. *Water*, 12(10), 2020.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[11] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *arXiv preprint arXiv:2107.14171*, 2021.

[12] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. Action branching architectures for deep reinforcement learning. *CoRR*, abs/1711.08946, 2017.

[13] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[14] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[15] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, Qiaobo Chen, Yinyuting Yin, Hao Zhang, Tengfei Shi, Liang Wang, Qiang Fu, Wei Yang, and Lanxiao Huang. Mastering complex control in MOBA games with deep reinforcement learning. *CoRR*, abs/1912.09729, 2019.

[16] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *CoRR*, abs/2006.14171, 2020.

[17] Adil Zouitine. Masking in deep reinforcement learning, Feb 2021.

[18] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C. Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *CoRR*, abs/2108.13264, 2021.

[19] Zhengfu Rao and Fernando Alvarruiz. Use of an artificial neural network to capture the domain knowledge of a conventional hydraulic simulation model. *Journal of Hydroinformatics*, 9(1):15–24, 01 2007.

[20] Gergely Hajgató, Bálint Gyires-Tóth, and György Paál. Reconstructing nodal pressures in water distribution systems with graph neural networks. *CoRR*, abs/2104.13619, 2021.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[22] John Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9, 1996.

[23] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[24] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

A DQN: Deep Q-Network algorithm

One of the most significant advancements in reinforcement learning was the development of the deep Q network algorithm, Mnih et al. [21]. It adds the power of deep neural networks to Q-learning. In fact, Q-Learning has been proven to converge towards the optimal solution in a tabular case or using linear function approximation. However, large and continuous state and action spaces need non-linear function approximators such as a neural network, and it is known that Q-learning is unstable or even diverges when using such an approximator [22]. With the advances in training deep neural networks, Mnih et al. in their paper *Human-level control through deep reinforcement learning* [21] addressed this problem and reconciled Q-learning with deep neural networks. Thus, giving birth to DQN and igniting research in reinforcement learning.

A.1 DQN

DQN is the first scalable reinforcement learning algorithm that combines successfully Q-learning with deep neural networks. To overcome stability issues, DQN adopts the following new techniques that turned out to be essential for the balance of the algorithm.

A.1.1 Replay memory

To use more IID data during SGD iterations, DQN introduced a replay memory (also called experienced replay) to collect and store the experience in a large buffer. This buffer ideally contains as many experiences as possible. When doing SGD, a random mini-batch will be gathered from the experienced replay and used in the optimization procedure. Since the replay memory buffer holds varied experiences, the mini-batch that's sampled from it will be diverse enough to provide independent samples. Another very important feature behind the use of an experience replay is that it enables the reusability of the data as the experiences will be sampled multiple times. This greatly increases the data efficiency of the algorithm compared to on-policy algorithms.

A.1.2 Target network

The moving target problem is due to continuously updating the network during training, which also modifies the target values. Nevertheless, the neural network has to update itself in order to provide the best possible state-action values. The solution that's employed in DQNs is to use two neural networks. One is called the online network, which is constantly updated, while the other is called the target network, which is updated only every N iteration (with N usually being between 1,000 and 10,000). The online network is used to interact with the environment while the target network is used to predict the target values. In this way, for N iterations, the target values that are produced by the target network remain fixed, preventing the propagation of instabilities and decreasing the risk of divergence. A potential disadvantage is that the target network is an old version of the online network. Nonetheless, in practice, the advantages greatly outweigh the disadvantages and the stability of the algorithm improves significantly.

A.1.3 The DQN algorithm

With the further employment of the separate Q-target network Q' with weight θ', and the learning update applied to mini-batches drawn uniformly from the experienced buffer. The loss function is expressed as:

$$L(\theta) = E_{(s,a,r,s')\sim\mathcal{D}}\big[\big(r + \max_{a'} Q'_{\theta'}(s',a') - Q_\theta(s,a)\big)^2\big] \tag{3}$$

An improvement over was introduced by Van Hasselt, Guez, and Silver in [23], to address the over-optimism in the Q-value estimations by using the current Q-network to select the next greedy action, but evaluating it using the target network. Equation 3), then become:

$$L(\theta) = E_{(s,a,r,s')\sim\mathcal{D}}\big[\big(r + Q'_{\theta'}(s,\ \operatorname*{argmax}_{a'} Q_\theta(s',a')) - Q_\theta(s,a)\big)^2\big] \tag{4}$$

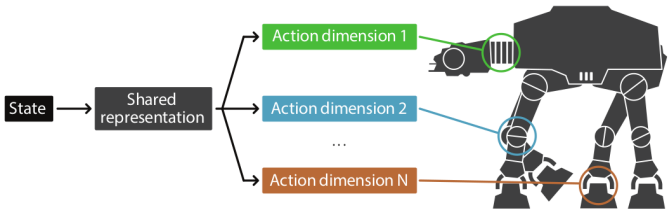


Figure 7: From [12] A conceptual illustration of the proposed action branching network architecture. The shared network module computes a latent representation of the input state that is then passed forward to the several action branches. Each action branch is responsible for controlling an individual degree of freedom and the concatenation of the selected sub-actions results in a joint-action tuple.

A.1.4 The curse of dimensionality in the DQN algorithm

Despite the success of the DQN algorithm which has enabled the use of reinforcement learning in domains with unprocessed, high-dimensional input. Its application to high-dimensional action spaces continues to suffer from the same issues as in tabular reinforcement learning, that is, the number of actions that need to be explicitly represented grows exponentially with increasing action dimensionality.

This is the case of our WDS environments, where the number of actions grows exponentially with the number of valves. This becomes a serious drawback of the DQN algorithm for a number of valves as small as 20, which corresponds to a number of actions of 1 048 576 (for discrete actions), and therefore needs a NN with 1 048 576 output cells. This renders the DQN algorithm impractical for large numbers of valves or for continuous actions.

B BDQ: Branching dueling Q-Network

In [12] A. Tavakoli, F. Pardo, and P. Kormushev introduced a novel neural architecture that enables the use of discrete-action algorithms in deep reinforcement learning for domains with high-dimensional discrete or continuous action spaces. The core notion of the proposed architecture is to distribute the representation of the action controllers across individual network branches, meanwhile, maintaining a shared decision module among them to encode a latent representation of the input and help with the coordination of the branches (see Figure 7). The proposed decomposition of the actions enables the linear growth of the total number of network outputs with increasing action dimensionality as opposed to the combinatorial growth in discrete-action algorithms such as DQN.

To demonstrate the capabilities of this new network architecture, the authors used it in implementing a new variant of the dueling DQN algorithm called Branching Dueling Q-network (BDQ). Which is, as it name suggests, an adaptation of the dueling DQN with the branching architecture. They were able to achieve state-of-the-art results on many challenging benchmarks where other discrete action algorithms have failed.

B.1 Action branching architecture

The key insight behind the proposed action branching architecture is that for solving problems in multidimensional action spaces, it is possible to optimize for each action dimension with a degree of independence. However, it is well-known that the naive distribution of the value function or the policy representation across several independent function approximators is subject to numerous challenges which can lead to convergence problems [12]. To address this, the proposed neural architecture distributes the representation of the value function or the policy across several network branches while keeping a shared decision module among them to encode a latent representation of the common input state (see Figure 7). This kind of architecture can also be found in nature. Octopuses, for instance, have complex neural systems where each arm is able to function with a degree of autonomy and even respond to stimuli after being detached from the central control.

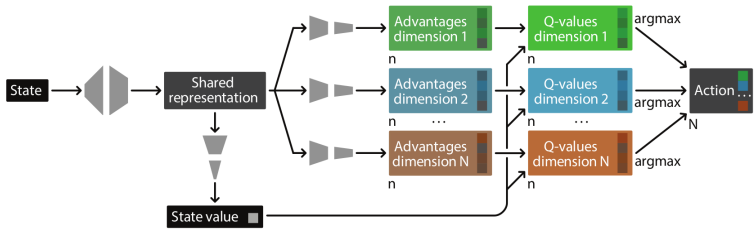


Figure 8: From [12] action branching network used in the BDQ algorithm.

B.2 BDQ

Besides the improvements on the vanilla DQN algorithm we presented earlier, two more extensions were incorporated into the BDQ agent:

- The dueling architecture
- The prioritized replay buffer

B.2.1 Dueling

For some states, different actions are not relevant to the expected value, and we do not need to learn the effect of each action for such states. So decoupling the action-independent value of state and Q-value may lead to more robust learning. Dueling DQN proposes a new network architecture to achieve this idea (Wang et al. [24]). More precisely, the Q-value can be split into the state value part and action advantage part as follows:

$$Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a) \tag{5}$$

and dueling DQN separates the representations of these two parts by

$$Q_\pi(s,a;\theta,\theta_V,\theta_A) = V_\pi(s;\theta,\theta_V) + \left(A_\pi(s,a;\theta,\theta_A) - \frac{1}{|\mathcal{A}|}\sum_{a'} A_\pi(s,a';\theta,\theta_A)\right) \tag{6}$$

Where θ_V and θ_A are parameters of two streams of layers, one for the state value $V_\pi(s)$ and the other for the state-action advantage $A_\pi(s,a)$. Subtracting the mean of the advantages from each individual advantage and then summing them with the state value results in improved learning stability when compared to the naive summation of the state value and advantages.

The dueling network architecture has been shown to lead to better policy evaluation in the presence of many similar-valued (or redundant) actions, and thus achieves faster generalization over large action spaces.

B.3 Prioritized experience replay

Instead of sampling uniformly from the replay buffer, PER samples more frequently transition with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error.

B.3.1 Adapting the DQN algorithm

Common state-value estimator As demonstrated in the action branching network of Figure 8, to adapt the dueling architecture into the action branching network, the representation of the (state-dependent) action advantages is distributed on the several action branches, meanwhile, adding a single additional branch for estimating the state-value function. Similar to the dueling architecture, the advantages and the state value are combined, via a special aggregating layer, to produce estimates of the distributed action values. The best performing method is to locally subtract each branch's mean advantage from its sub-action advantages, prior to their summation with the state value.

Formally, for an action dimension $d \in \{1, ..., N\}$ with $|\mathcal{A}_d| = n$ discrete sub-actions, the individual branch's Q-value at state s and sub-action a_d is expressed in terms of the common state value V(s) and the corresponding (state-dependent) sub-action advantage $A_d(s, a_d)$ by:

$$Q_d(s, a_d) = V(s) + \big(A_d(s, a_d) - \frac{1}{n} \sum_{a'_d \in \mathcal{A}_d} A_d(s, a'_d)\big) \tag{7}$$

Temporal-difference target Three different methods were proposed by the paper [12]. The first approach is to calculate a TD target, for each individual action dimension separately. The second is to consider the same target for all the branches, by taking the maximum over the branches. And finally, take the mean over the branches instead of the maximum for a common target for all the branches. They reported the third option to be the best in their experiments:

$$y = r + \gamma \, \frac{1}{N} \sum_{d} Q'_d(s, \ \operatorname*{argmax}_{a' \in \mathcal{A}_d} Q_\theta(s', a')) \tag{8}$$

Loss function To avoid the canceling out of the branches errors of different signs, the loss function is defined as the mean squared TD error across the branches:

$$L = E_{(s,a,r,s') \sim \mathcal{D}} \big[\sum_{d} \big(y_d - Q_d(s, a_d)\big)^2 \big] \tag{9}$$

where $\mathcal{D}$ denotes a (prioritized) experience replay buffer and a denotes the joint-action tuple $(a_1, a_2, ..., a_N)$.

Error for experience prioritization A simple way of aggregating the branches' TD errors is to take the unified prioritization error to be the sum of a transition's absolute, distributed TD errors:

$$e_{\mathcal{D}}(s, a, r, s') = \sum_{d} |y_d - Q_d(s, a_d)| \tag{10}$$

C Hyper-parameters

C.1 DQN

Hyper-parameter	Value
hidden-sizes	[512, 256]
epsilon-train	0.73
epsilon-test	0.01
epsilon-decay	5×10^-6
buffer-size	10^5
learning-rate	$8e - 5$
gamma	0.99
target-update-freq	500
epoch	1000
step-per-epoch	8×10^4
step-per-collect	24
update-per-step	1/24
batch-size	128
training-environments-number	24
test-environments-number	1

C.2 BDQ and BDQF

Hyper-parameter	Value
common-hidden-sizes	[512, 256]
action-hidden-sizes	128
value-hidden-sizes	128
epsilon-train	0.73
epsilon-test	0.01
epsilon-decay	5×10^-6
buffer-size	10^5
learning-rate	$8e-5$
gamma	0.99
target-update-freq	500
epoch	1000
step-per-epoch	8×10^4
step-per-collect	24
update-per-step	1/24
batch-size	128
training-environments-number	24
test-environments-number	1

D Compute resources

The runs were performed on the CTE-power9 cluster of Barcelona Supercomputing Center, each run was using one Tesla V100 GPU and 40 CPUs to run 24 Epanet environments in parallel as well as the learning algorithm. There were at most 10 running at the same time. The run-time was on the horizon of 48hours.

Algorithm 1: BDQF (BDQ for Fixed actions)

Hyperparameters: replay buffer capacity M, reward discount factor γ, delayed steps C for target network update, ϵ-greedy factor ϵ, number of steps T for testing the model
Inputs: number of episodes N, environment Env takes in the current state-action pair and outputs the reward and next state
Initialize parameters θ of action-value function Q
Initialize target network Q' with parameters $\theta' \leftarrow \theta$
Fill the replay buffer $\mathcal{D}$
For episode = 0,1,2,...,N **do**
Initialize environment with leak, get fixed actions a_f and initial state s_0
  For step t = 0,1,2... **do**
    With probability ϵ select a random action vector a_t, otherwise select $a_t = \left(\underset{a_d \in \mathcal{A}_d}{argmax}\ Q_\theta(s_t, a_d)\right)$
    $r_t, s_{t+1} \leftarrow Env(s_t, a_t)$
    If the episode has ended, set $f_t = 1$. Otherwise set $f_t = 0$
    Store experience $E = (a_t, a_t, r_t, f_t, s_{t+1})$ in $\mathcal{D}$ FIFO
    Sample prioritized minibatch of transitions $E = (s_i, a_i, r_i, f_i, s_{i+1})$ from $\mathcal{D}$
    If $f_i = 0$, set the common target $y_i = r_i + \gamma \frac{1}{N_{nf}} \sum_{d_{nf}} Q'_d(s, \underset{a' \in \mathcal{A}_{d_{nf}}}{\text{argmax}}\ Q_\theta(s', a'))$. Otherwise, set $y_i = r_i$
    Perform a gradient descent step on $\sum_{d_{nf}} \left(y_{i_{d_{nf}}} - Q_{d_{nf}}(s_i, a_{i_{d_{nf}}})\right)^2$
    Synchronize the target Q' every C steps
    Evaluate and save the model every T steps
    If the episode has ended, break the loop
  End for
End for

E NeurIPS questions

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? **[TODO]**yes
   (b) Did you describe the limitations of your work? **[TODO]**NA
   (c) Did you discuss any potential negative social impacts of your work? **[TODO]**NA
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? **[TODO]**yes
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? **[TODO]**yes
   (b) Did you include complete proofs of all theoretical results? **[TODO]**yes
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? **[TODO]**No, the data is proprietary
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? **[TODO]**yes
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? **[TODO]**yes
   (d) Did you include the total amount of computing and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? **[TODO]**yes
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? **[TODO]**yes
   (b) Did you mention the license of the assets? **[TODO]**yes

(c) Did you include any new assets either in the supplemental material or as a URL? **[TODO]**NA

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? **[TODO]**yes

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? **[TODO]**NA

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? **[TODO]**NA

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? **[TODO]**NA

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? **[TODO]**NA